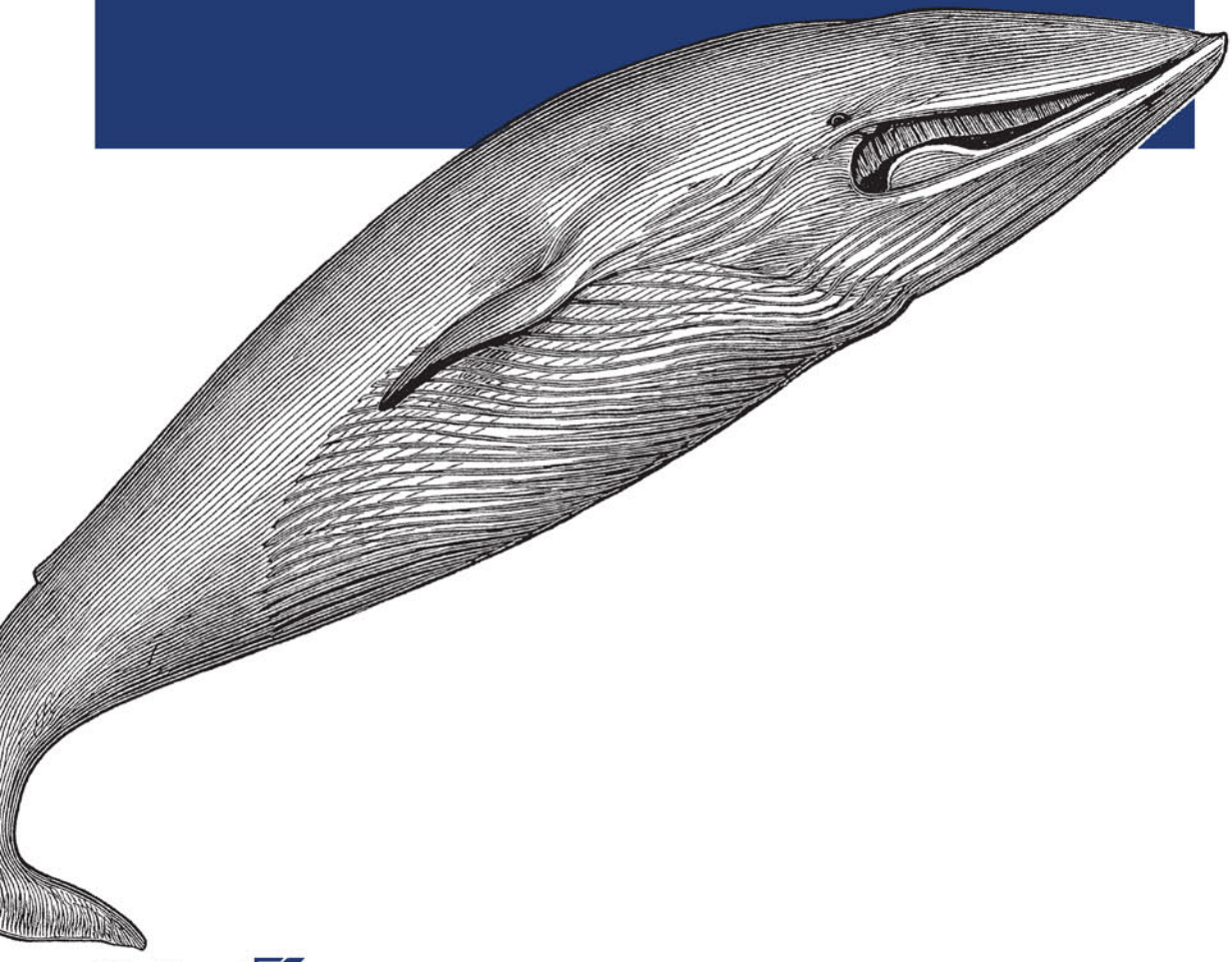
O'REILLY®

Docker

Praktyczne zastosowania



Helion

Karl Matthias, Sean P. Kane

Tytuł oryginału: Docker: Up and Running

Tłumaczenie: Andrzej Stefański

ISBN: 978-83-283-2904-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/docker
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Przedmowa **11**

Wstęp **15**

1. Wprowadzenie **19**
Narodziny Dockera 19
Co obiecuje Docker 19
Korzyści płynące ze stosowania procesów proponowanych przez Dockera 21
Czym Docker nie jest 23

2. Rzut oka na Dockera **25**
Upraszczanie procesów 25
Duże wsparcie i wykorzystanie 27
Architektura 28
Model klient-serwer 28
Porty sieciowe i gniazdka sieciowe 29
Rozbudowane narzędzia 29
Tekstowy klient Dockera 30
API 30
Sieć w kontenerze 31
Najlepsze zastosowania Dockera 32
Kontenery to nie maszyny wirtualne 33
Kontenery są lekkie 33
Dążenie do niezmienności infrastruktury 33
Ograniczona izolacja 34
Aplikacje bezstanowe 34
Przenoszenie informacji o stanie na zewnątrz 35

Schemat pracy z Dockerem 36
Wersjonowanie 36
Budowanie 37
Testowanie 38
Tworzenie pakietów 38
Wdrażanie 39
Ekosystem Dockera 39
Podsumowanie 41

3. Instalacja Dockera ... 43
Ważne pojęcia 43
Klient Dockera 44
Linux 45
Mac OS X 10.10 47
Microsoft Windows 8 48
Serwer Dockera 48
Linux korzystający z systemd 49
Linux wykorzystujący upstart 49
Linux wykorzystujący init.d 49
Serwery na maszynach wirtualnych 50
Testowanie 58
Ubuntu 58
Fedora 58
CentOS 58
Podsumowanie 59

4. Praca z obrazami Dockera ... 61
Anatomia pliku Dockerfile 61
Budowanie obrazu 64
Uruchamianie zbudowanego obrazu 68
Zmienne środowiska 69
Własne obrazy bazowe 69
Zapisywanie obrazów 70
Publiczne rejestry 70
Rejestry prywatne 71
Autoryzacja w rejestrze 71
Tworzenie kopii rejestru 74
Inne sposoby dostarczania obrazów 77

5. Praca z kontenerami Dockera 79
Czym jest kontener? 79
Historia kontenerów 80
Tworzenie kontenera 81
Podstawowa konfiguracja 82
Magazyny danych 85
Ograniczenia zasobów 87
Uruchamianie kontenera 92
Automatyczne restartowanie kontenera 93
Zatrzymywanie kontenera 94
Wymuszanie zakończenia pracy kontenera 95
Pauzowanie i wznawianie pracy kontenera 96
Czyszczenie kontenerów i obrazów 96
Kolejne kroki 98

6. Poznawanie Dockera 99
Wyświetlanie wersji Dockera 99
Informacje o serwerze 100
Pobieranie aktualizacji obrazów 101
Pobieranie informacji o kontenerze 102
Wnętrze działającego kontenera 103
docker exec 103
nsenter 104
Badanie powłoki 107
Zwracanie wyniku 107
Logi Dockera 109
Monitorowanie Dockera 112
Statystyki kontenerów 112
docker events 115
cAdvisor 116
Dalsze eksperymenty 120

7. Tworzenie kontenerów produkcyjnych 121
Wdrażanie 121
Klasy narzędzi 122
Narzędzia do koordynacji 123
Narzędzia do planowania przetwarzania rozproszonego 123
Podsumowanie 125
Testowanie kontenerów 125
Szybki przegląd 125
Zewnętrzne zależności 128

8. Debugowanie kontenerów 129
Dane generowane przez proces 129
Przeglądanie procesów 133
Kontrolowanie procesów 134
Przeglądanie sieci 135
Historia obrazów 136
Przeglądanie kontenera 136
Przeglądanie systemu plików 138
Dalsze kroki 138

9. Skalowanie Dockera 139
Docker Swarm 140
Centurion 144
Amazon EC2 Container Service 148
Konfiguracja IAM 148
Przygotowanie AWS CLI 149
Instancje kontenerów 150
Zadania 153
Testowanie zadania 157
Zatrzymywanie zadania 158
Podsumowanie 159

10. Zagadnienia zaawansowane 161
Mechanizmy wymienne 161
Sterownik uruchamiania 161
Magazyny danych 163
Szczegółowo o kontenerach 166
Grupy kontrolne (cgroups) 166
Przestrzeń nazw jądra, przestrzeń nazw użytkownika 169
Bezpieczeństwo 173
Czy kontener jest bezpieczny? 173
Czy demon Dockera jest bezpieczny? 178
Sieć 180

11. Projektowanie produkcyjnej platformy dla kontenerów 185
12factor 186
Repozytorium kodów 186
Zależności 186
Konfiguracja 188
Usługi pomocnicze 190
Budowanie, udostępnianie, uruchamianie 190

Procesy 190
Wykorzystanie portów 191
Współbieżność 191
Dyspozycyjność 192
Podobieństwo środowiska programistycznego i produkcyjnego 192
Logi 193
Procesy administracyjne 193
Podsumowanie twelve-factor 194
The Reactive Manifesto 194
Responsywność 194
Stabilność 194
Elastyczność 194
Obsługa komunikatów 195
Podsumowując 195

12. Wnioski 197
Wyzwania 197
Przepływ pracy w Dockerze 198
Minimalizowanie liczby obiektów do wdrożenia 198
Optymalizacja przechowywania i przesyłania danych 199
Korzyści 199
Słowo końcowe 200

Skorowidz 201

ROZDZIAŁ 2.

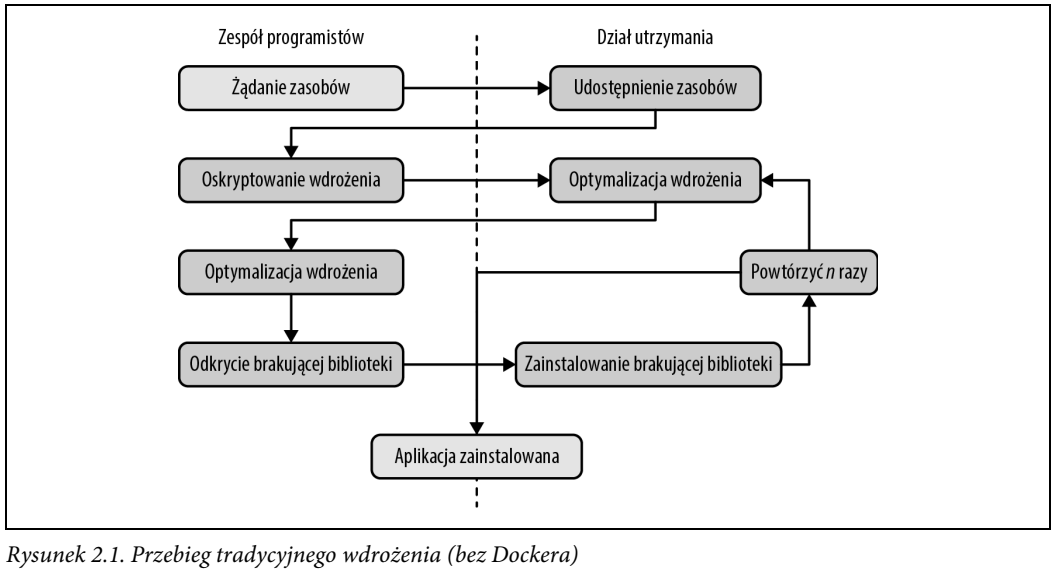
Rzut oka na Dockera

Zanim zagłębimy się w konfigurowanie i instalowanie Dockera, zaczniemy od szybkiego wprowadzenia, by wytłumaczyć, czym jest Docker i co wnosi. Jest to potężna technologia, ale niezbyt skomplikowana. W tym rozdziale omówimy ogólne podstawy działania Dockera, powiemy, co czyni go tak potężnym, i przedstawimy kilka powodów, by z niego skorzystać. Jeśli to czytasz, masz prawdopodobnie własne powody, by go wykorzystać, ale nie zaszkodzi poszerzyć swoją wiedzę przed zagłębieniem się w szczegóły.

Nie obawiaj się — nie zatrzyma Cię to na zbyt długo. W kolejnym rozdziale zajmiemy się od razu instalacją i uruchomieniem Dockera w Twoim systemie.

Upraszczanie procesów

Docker może uprościć zarówno przepływ pracy, jak i komunikację, ale zazwyczaj to tylko początek wdrożenia. Tradycyjny cykl produkcyjnego wdrażania aplikacji często wygląda podobnie do poniższego (pokazano go też na rysunku 2.1):



Rysunek 2.1. Przebieg tradycyjnego wdrożenia (bez Dockera)

1. Programiści tworzący aplikację zgłaszają zapotrzebowanie na zasoby od inżynierów odpowiedzialnych za utrzymanie.
2. Zasoby są dostarczane i przekazywane programistom.
3. Programiści tworzą skrypty i narzędzia do wdrożenia.
4. Inżynierowie odpowiedzialni za utrzymanie i programiści iteracyjnie wprowadzają poprawki w procesie wdrożenia.
5. Programiści odkrywają kolejne zależności aplikacji.
6. Inżynierowie odpowiedzialni za utrzymanie instalują wszystkie dodatkowe zasoby.
7. Kroki 5. i 6. powtarzane są *n* razy.
8. Aplikacja zostaje wdrożona.

Z naszego doświadczenia wynika, że wdrożenie nowej aplikacji w środowisku produkcyjnym może zająć w przypadku nowych, złożonych systemów większą część tygodnia. Nie jest to zbyt produktywne i nawet jeśli praktyki DevOps pozwalają zmniejszyć niektóre z tych barier, często wymagają wiele wysiłku i komunikacji między zespołami ludzi. Ten proces może być często wyzwaniem technicznym i nieść ze sobą duże koszty, a co gorsza, może ograniczać zakres innowacji, jakie zespół programistów będzie wprowadzał w przyszłości. Jeśli wdrożenie oprogramowania jest procesem trudnym, czasochłonnym i wymagającym zasobów innego zespołu, programiści często wbudowują wszystko w istniejącą aplikację, aby uniknąć trudności związanych z nowym wdrożeniem.

Systemy pozwalające na automatyzację wdrożenia, takie jak Heroku (*https://www.heroku.com/*), pokazały programistom, jak może wyglądać świat, jeśli masz kontrolę nad większością wykorzystywanych zasobów oraz nad samą aplikacją. Rozmowa z programistami na temat wdrożenia często kończy się dyskusją na temat tego, jaki ten świat jest prosty. Jeśli jesteś inżynierem odpowiedzialnym za utrzymanie, prawdopodobnie słyszałeś narzekania na to, o ile wolniejsze są Twoje wewnętrzne systemy w porównaniu z wdrażaniem z Heroku.

Docker nie próbuje być Heroku, ale umożliwia przejrzyste rozgraniczenie odpowiedzialności i dołączenie zależności, co w rezultacie daje podobny wzrost produktywności. Umożliwia też jeszcze dokładniejszą kontrolę niż Heroku, dając programistom kontrolę nad wszystkim aż do poziomu dystrybucji systemu operacyjnego, na którym działa dostarczana przez nich aplikacja.

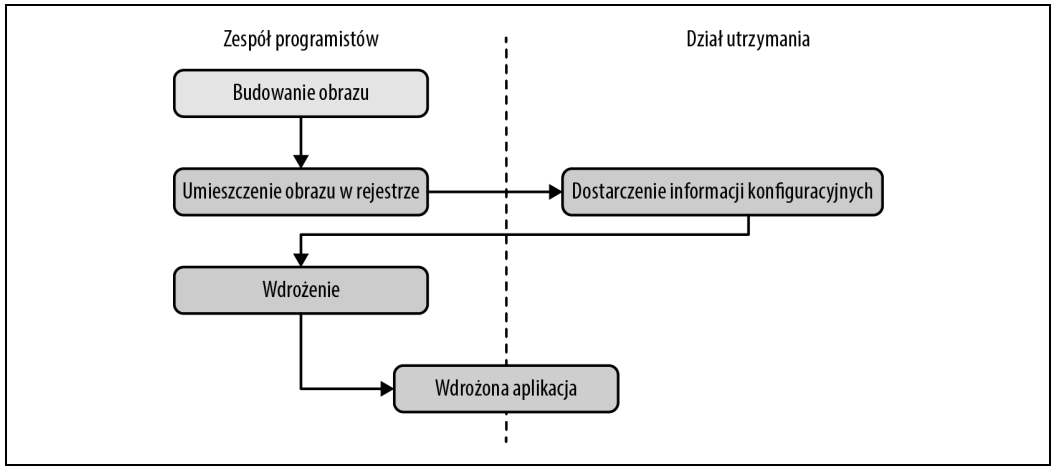
Jako firma Docker propaguje podejście „baterie dołączone, ale wymienne". Oznacza to, że stara się dostarczać swoje narzędzia wraz ze wszystkim, czego większość użytkowników będzie potrzebowała do ich użycia. Jednocześnie jednak poszczególne części można w łatwy sposób zamieniać, by dostosować je do konkretnych zastosowań.

Korzystając z repozytorium obrazów jako punktu wyjścia, Docker umożliwia oddzielenie odpowiedzialności za kompilację obrazu aplikacji od wdrożenia i utrzymania kontenera.

W praktyce oznacza to, że zespoły odpowiedzialne za programowanie mogą kompilować swoje aplikacje wraz ze wszystkimi zależnościami, uruchamiać je w środowiskach deweloperskich i testowych, a następnie umieszczać ten sam zestaw zawierający aplikację i jej zależności w środowisku produkcyjnym. Ponieważ wszystkie te pakiety z zewnątrz wyglądają tak samo, inżynierowie odpowiedzialni za utrzymanie mogą następnie utworzyć lub zainstalować standardowe narzędzia

umożliwiające wdrożenie i uruchomienie aplikacji. Cykl opisany na rysunku 2.1 dzięki temu wygląda podobnie do poniższego (pokazanego również na rysunku 2.2):

- Programiści przygotowują obraz Dockera i przekazują go do rejestru.
- Inżynierowie odpowiedzialni za utrzymanie dodają szczegóły konfiguracyjne do kontenera i przygotowują zasoby.
- Deweloperzy uruchamiają wdrożenie.



Rysunek 2.2. Proces wdrożenia z Dockerem

Jest to możliwe, ponieważ Docker pozwala wykryć wszystkie problemy z zależnościami podczas programowania i cykli testowych. Gdy aplikacja jest gotowa do pierwszego wdrożenia, ta praca jest już wykonana. Nie wymaga to też zazwyczaj tylu iteracji przekazywania informacji pomiędzy zespołami odpowiedzialnymi za programowanie i utrzymanie. Jest dużo prostsze i oszczędza wiele czasu. Co więcej, prowadzi do uzyskania oprogramowania o lepszej jakości dzięki testowaniu środowiska wdrożenia przed wydaniem kolejnej wersji.

Duże wsparcie i wykorzystanie

Docker jest coraz lepiej wspierany. Większość dużych publicznych chmur ogłosiła przynajmniej częściowe bezpośrednie jego wsparcie. Docker działa w AWS Elastic Beanstalk, Google AppEngine, IBM Cloud, Microsoft Azure, Rackspace Cloud i wielu innych chmurach. Na DockerCon 2014 Eric Brewer z Google ogłosił, że firma ta będzie wspierała Dockera jako swój podstawowy wewnętrzny format kontenerów. Dla społeczności Dockera oznacza to nie tylko dobry PR, ale przede wszystkim to, że wielkie pieniądze zaczną wspierać stabilność Dockera i przyczyniać się do jego sukcesu.

Kontenery Dockera umacniają swoją pozycję — stają się popularnym formatem wśród firm dostarczających platformy chmurowe i pozwalają tworzyć aplikacje chmurowe typu „napisz raz, uruchamiaj wszędzie”. Gdy Docker zaprezentował swoją bibliotekę `libswarm` na DockerCon 2014, inżynier z Orchard zademonstrował wdrożenie kontenera Dockera w heterogenicznym środowisku złożonym z wielu chmur. Tego rodzaju zarządzanie nie było dotąd tak łatwe i jest

prawdopodobne, że gdy największe firmy nadal będą inwestować w platformę, wsparcie i narzędzia będą stawały się coraz lepsze.

Powiedzieliśmy o kontenerach i narzędziach Dockera, ale co ze wsparciem producentów systemów operacyjnych i ich wykorzystaniem? Klient Dockera działa bezpośrednio w większości najważniejszych systemów operacyjnych, ale dlatego, że serwer Dockera korzysta z kontenerów linuksowych, nie uruchamia się w systemach innych niż Linux. Docker był tworzony w dystrybucji Ubuntu Linux, ale większość dystrybucji Linuksa i inne największe systemy operacyjne są teraz obsługiwane, jeśli tylko jest to możliwe.

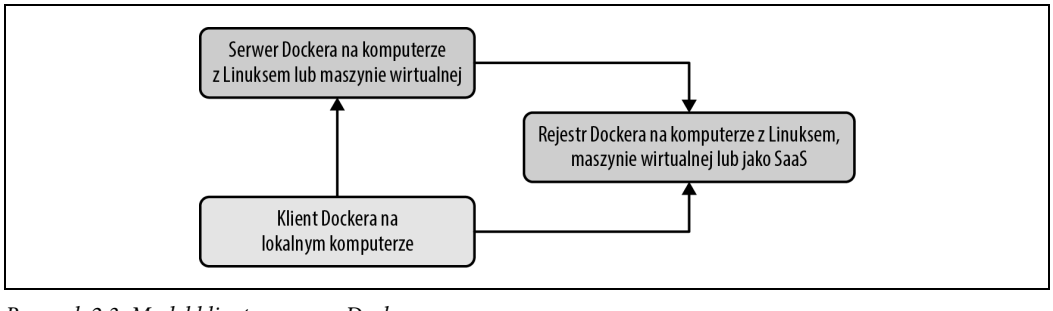
Gdy Docker miał zaledwie dwa lata, wspierał już wiele platform. Wraz z utworzeniem The Open Container Initiative (*https://www.opencontainers.org/*) w czerwcu 2015 roku zwiększyły się nadzieje na przygotowanie porządnego standardu branżowego dla kontenerów i dalszy rozwój Dockera.

Architektura

Docker jest potężną technologią, często oznaczającą coś bardzo skomplikowanego. Podstawowa architektura Dockera to jednak prosty model klient-serwer z jednym tylko modułem wykonywalnym, który obsługuje oba komponenty w zależności od tego, z jakimi parametrami zostanie uruchomiony. Pod tą prostą powłoką zewnętrzną Docker intensywnie wykorzystuje mechanizmy jądra takie jak `iptables`, wirtualne mostkowanie, `cgroups`, przestrzenie nazw i różne sterowniki systemów plików. Niektóre z nich omówimy w rozdziale 10. Na razie skupimy się na tym, w jaki sposób działają klient i serwer oraz krótko omówimy warstwę sieciową znajdującą się w kontenerze Dockera.

Model klient-serwer

Docker składa się z co najmniej dwóch części: klienta oraz serwera-demona (patrz rysunek 2.3). Opcjonalnie pojawia się trzeci komponent, nazywany rejestrem, który przechowuje obrazy Dockera i metadane opisujące te obrazy. Serwer zajmuje się uruchamianiem kontenerów i zarządzaniem nimi, a z klienta korzysta się, by przekazać serwerowi, co jest do zrobienia. Demon Dockera (*http://bit.ly/2dGp26h*) może działać na dowolnej liczbie serwerów należących do infrastruktury, a pojedynczy klient może komunikować się z dowolną liczbą serwerów. Całą komunikację inicjują klienci, a serwery Dockera mogą kierować się bezpośrednio do rejestrów z obrazami, jeśli dostaną od klienta takie polecenie. Klienci są odpowiedzialni za wyznaczanie serwerom, co jest do zrobienia, a serwery skupiają się na utrzymywaniu kontenerów z aplikacjami.



Rysunek 2.3. Model klient-serwer w Dockerze

Struktura Dockera odrobinę różni się od niektórych innych rozwiązań typu klient-serwer. Zamiast oddzielnych modułów wykonywalnych klienta i serwera wykorzystywany jest ten sam kod dla obu komponentów. Instalując Dockera, otrzymujesz oba komponenty, ale serwer uruchomi się jedynie w systemie Linux. Uruchomienie serwera-demona Dockera wymaga jedynie wywołania Dockera z parametrem `-d`, który sprawia, że uruchomiony program pracuje jako demon i nasłuchuje w oczekiwaniu na przychodzące połączenia. Każdy komputer z Dockerem powinien mieć zazwyczaj jeden działający demon Dockera, który może zarządzać wieloma kontenerami. W takiej sytuacji można korzystać z działającego w trybie tekstowym klienta do komunikacji z demonem.

Porty sieciowe i gniazdka sieciowe

Działające w trybie tekstowym narzędzie oraz demon wywoływany za pomocą polecenia `docker -d` porozumiewają się poprzez gniazdka sieciowe. Możesz skonfigurować demon Dockera w taki sposób, by nasłuchiwał na jednym gniazdku TCP lub Unix albo większej liczbie gniazdek. Można na przykład sprawić, by Docker nasłuchiwał zarówno na lokalnym gniazdku Unix, jak i dwóch różnych portach TCP (szyfrowanym i nieszyfrowanym). W wielu dystrybucjach systemu Linux jest to ustawienie domyślne. Jeśli chcesz ograniczyć możliwość dostępu do lokalnego systemu, najbezpieczniejszą opcją będzie nasłuchiwanie jedynie na gniazdku Unix. Większość użytkowników chce jednak komunikować się z demonem Dockera zdalnie, dlatego najczęściej nasłuchuje on przynajmniej na jednym porcie TCP.

Pierwotnie Docker nasłuchiwał na porcie 4243, ale port ten nigdy nie został zarejestrowany, a w zasadzie był już wykorzystywany przez inne narzędzie — program do tworzenia kopii zapasowych CrashPlan, działający w systemie Mac OS X. W efekcie dla Dockera został zarejestrowany w IANA inny port TCP i obecnie zazwyczaj jest on konfigurowany w taki sposób, że korzysta z portu 2375 TCP do komunikacji nieszyfrowanej i 2376 — do obsługi ruchu szyfrowanego.

W Dockerze 1.3 i późniejszych domyślnie wykorzystywany jest szyfrowany port 2376, ale łatwo zmienić tę konfigurację. Gniazdko Unix znajduje się w różnych miejscach w zależności od systemu operacyjnego, dlatego musisz sprawdzić, gdzie umieszczone jest w Twoim przypadku. Jeśli bardzo Ci na tym zależy, możesz łatwo to zmienić podczas instalacji. W przeciwnym razie powinna się sprawdzić wartość domyślna.

Rozbudowane narzędzia

Jednym z wielu czynników, jakie doprowadziły do rosnącego wykorzystania Dockera, jest jego prosty, ale potężny zestaw narzędzi. Zestaw ten ciągle się powiększa od chwili udostępnienia Dockera i w dużej mierze dzieje się tak dzięki społeczności skupionej wokół Dockera. Narzędzia dostarczane z Dockerem służą do tworzenia jego obrazów i ich wdrażania na lokalnych serwerach Dockera, ale też zawierają wszystkie mechanizmy niezbędne do zarządzania zdalnym serwerem Dockera. Wysiłek społeczności skupił się na zarządzaniu całymi flotami (czy klastrami) serwerów Dockera oraz na planowaniu i koordynowaniu wdrażania kontenerów. Twórcy Dockera również pracują nad zestawem tego typu narzędzi, w skład których wchodzą: Compose (*https://github.com/docker/compose*) (wcześniej znany jako Fig), Machine (*https://github.com/docker/machine*) oraz Swarm (*https://github.com/docker/swarm/*), których zadaniem jest umożliwienie wykonania spójnego wdrożenia w różnych środowiskach.

Ponieważ Docker dostarcza zarówno narzędzie pracujące w trybie tekstowym, jak i dostępne zdalnie API, tworzenie narzędzi w dowolnym języku jest proste. Narzędzie działające w trybie tekstowym dobrze nadaje się do tworzenia skryptów, dzięki czemu za pomocą prostych skryptów powłoki korzystających z tego narzędzia można sporo osiągnąć.

Tekstowy klient Dockera

Narzędzie Dockera pracujące w trybie tekstowym jest jego najważniejszym interfejsem, z jakim większość użytkowników będzie miała do czynienia. Jest to napisany w języku Go (*https://golang.org/*) program, który kompiluje się we wszystkich popularnych systemach operacyjnych i na wszystkich architekturach procesorów. Narzędzie to jest częścią podstawowej dystrybucji Dockera na różnych platformach i również kompiluje się bezpośrednio ze źródeł Go. Wśród czynności, które można wykonać za pomocą tego narzędzia, są między innymi:

- tworzenie obrazów kontenera,
- pobieranie obrazów z rejestru do demona Dockera i wysyłanie ich do rejestru z demona Dockera,
- uruchamianie kontenera na serwerze Dockera na pierwszym planie lub w tle,
- pobieranie logów Dockera ze zdalnego serwera,
- uruchamianie wiersza poleceń wewnątrz kontenera działającego na zdalnym serwerze.

Jak widać, można to wykorzystać w procesie tworzenia i wdrażania. Jednak użycie narzędzia tekstowego Dockera nie jest jedynym sposobem komunikowania się z Dockerem, a nawet nie jest to narzędzie dające największe możliwości.

API

Tak jak wiele innych nowoczesnych aplikacji, demon Dockera ma dostępne zdalnie API (ang. *Application Programming Interface*). W rzeczywistości korzysta z niego też narzędzie tekstowe Dockera do komunikacji z demonem. Ponieważ API jest udokumentowane i dostępne publicznie, często bezpośrednio z niego korzystają narzędzia zewnętrzne. Umożliwia ono tworzenie narzędzi przeznaczonych do różnych celów — od zarządzania wdrożonymi kontenerami na serwerach, przez automatyzację wdrożeń, do rozproszonego planowania. Choć jest bardzo prawdopodobne, że początkujący użytkownicy nie będą chcieli zaczynać od bezpośredniego wykorzystania API Dockera, warto wiedzieć o jego istnieniu. Zwiększając z czasem zakres stosowania Dockera w swojej organizacji, prawdopodobnie zaczniesz coraz bardziej przekonywać się do tego, że API jest dobrym sposobem na integrację z tym narzędziem.

Dokładna dokumentacja tego API znajduje się na stronie internetowej Dockera (*https://docs.docker.com/reference/api/docker_remote_api/*). Wraz z dojrzewaniem całego ekosystemu zaczęły się pojawiać coraz porządniejsze implementacje bibliotek Docker API dla wielu popularnych języków programowania. My na przykład korzystaliśmy z bibliotek do języków Go oraz Ruby i okazało się, że są zarówno porządnie napisane, jak i szybko aktualizowane, gdy pojawiają się nowe wersje Dockera.

Większość czynności, jakie możesz wykonywać za pomocą narzędzi tekstowych Dockera, można stosunkowo prosto wykonać też za pomocą API. Dwoma ważnymi wyjątkami są przypadki wymagające strumieniowania lub dostępu do terminala: uruchamianie zdalnego wiersza poleceń i uruchamianie kontenera w trybie interaktywnym. W tych przypadkach często prościej jest korzystać z narzędzia wiersza poleceń.

Sieć w kontenerze

Choć kontenery Dockera składają się w zdecydowanej większości z procesów działających w samym systemie hosta, zachowują się one zupełnie inaczej niż inne procesy w warstwie sieci. Jeśli przyjmiesz, że każdy kontener Dockera zachowuje się w sieci jak komputer w prywatnej sieci lokalnej, będziesz na właściwej drodze. Serwer Dockera działa jako wirtualny most sieciowy, a kontenery są klientami połączonymi za jego pomocą. Most sieciowy to urządzenie, które przekazuje ruch z jednej strony na drugą. Możesz przyjąć, że jest to miniaturowa wirtualna sieć z podłączonymi do niej komputerami.

Zostało to zaimplementowane w taki sposób, że każdy kontener ma swój własny interfejs sieciowy podłączony do mostka sieciowego Dockera z własnym adresem IP przypisanym do wirtualnego interfejsu. Docker pozwala łączyć porty serwera z portami kontenera w taki sposób, by kontener był dostępny z zewnątrz. Taki ruch przed dotarciem do kontenera przechodzi przez mechanizm proxy, który również jest częścią serwera Dockera. Bardziej szczegółowe informacje możesz znaleźć w rozdziale 10.

Docker tworzy prywatną podsieć, wykorzystując wolny blok podsieci prywatnej według RFC 1918 (*https://tools.ietf.org/html/rfc1918*). Podczas startu sprawdza, które bloki sieciowe nie są wykorzystywane, po czym jeden z nich przypisuje do swojej sieci wirtualnej. Jest to połączone mostem z siecią lokalną hosta za pomocą dostępnego na serwerze interfejsu `docker0`. Oznacza to, że wszystkie kontenery znajdują się w jednej sieci i mogą się ze sobą bezpośrednio porozumiewać. Aby jednak dostać się do hosta lub wyjść na zewnątrz, muszą one przejść przez interfejs wirtualnego mostu `docker0`. Jak już wspomnieliśmy, ruch przychodzący przechodzi przez mechanizm proxy. Ma on stosunkowo dużą wydajność, ale może się okazać ograniczeniem, jeśli uruchomisz w kontenerach aplikacje przesyłające dużo danych. Więcej na ten temat powiemy podczas omawiania innych zagadnień sieciowych w rozdziale 10., gdzie zaproponujemy również kilka rozwiązań.

Liczba sposobów, na jakie można skonfigurować warstwę sieciową Dockera, jest zdumiewająca. Obejmuje mechanizmy począwszy od alokowania własnych segmentów sieciowych po konfigurowanie własnych, dopasowanych mostów sieciowych. Użytkownicy często korzystają z mechanizmów stosowanych domyślnie, ale czasem zdarza się, że potrzebne jest coś bardziej złożonego lub dopasowanego do konkretnego zastosowania. Dużo więcej szczegółowych informacji na temat mechanizmów sieciowych Dockera można znaleźć w jego dokumentacji (*https://docs.docker.com/engine/userguide/networking/*), a my z kolei podamy więcej szczegółów na temat sieci w rozdziale omawiającym zaawansowane zagadnienia.

Przygotowując swoje procesy wykorzystujące Dockera, powinieneś koniecznie zacząć od zastosowania domyślnych mechanizmów sieciowych. Może się potem okazać, że te domyślne mechanizmy są zbędne. Można wszystkie je wyłączyć za pomocą jednego parametru podczas uruchamiania demona Dockera. Nie da się ich konfigurować dla każdego kontenera z osobna, ale możesz je wyłączyć dla wszystkich kontenerów za pomocą parametru `--net host` przekazanego do polecenia `docker -d`. Gdy Docker jest uruchomiony w tym trybie, kontenery korzystają po prostu z urządzeń sieciowych i adresów hosta.

Najlepsze zastosowania Dockera

Tak jak większość narzędzi, Docker wspaniale sprawdza się w wielu zastosowaniach, ale w innych nie jest już tak idealny. Możesz na przykład otwierać słoik za pomocą młotka. Ma to jednak pewne wady. Zrozumienie, w jaki sposób najlepiej wykorzystać dane narzędzie, lub przynajmniej po prostu ustalenie, czy jest to odpowiednie narzędzie, może naprowadzić Cię na właściwą ścieżkę dużo szybciej.

Zacznijmy od tego, że architektura Dockera idealnie sprawdza się w przypadku aplikacji, które są bezstanowe albo w których informacja o stanie jest przechowywana w zewnętrznych magazynach danych takich jak bazy danych lub pamięć podręczna. Docker wymusza stosowanie pewnych dobrych praktyk programistycznych dla tej klasy aplikacji — o tym, dlaczego jest to tak ważne, powiemy później. Oznacza to jednak, że takie pomysły, jak umieszczanie bazy danych wewnątrz kontenera Dockera, są po prostu próbą płynięcia pod prąd. Nie chodzi o to, że nie możesz albo nie powinieneś tego zrobić — po prostu nie jest to najbardziej oczywisty sposób wykorzystania Dockera i jeśli od tego zaczniesz, szybko możesz się zniechęcić. Do dobrych zastosowań dla Dockera zaliczamy interfejsy sieciowe, API oraz wykonywanie krótko trwających czynności, takich jak uruchamianie skryptów administracyjnych, które mogą być wywoływane przez cron.

Tradycyjne aplikacje są w większości przypadków stanowe, co oznacza, że przechowują ważne dane w pamięci, plikach lub bazie danych. Jeśli zrestartujesz usługę stanową, możesz utracić pochodzące z pamięci informacje, które nie zostały jeszcze zapisane na nośniku trwałym. Z drugiej strony bezstanowe aplikacje są zazwyczaj zaprojektowane w taki sposób, by bezzwłocznie odpowiadać na pojedyncze zapytania, i nie muszą przechowywać informacji pomiędzy zapytaniami od jednego lub większej liczby klientów.

Jeśli na początku skupisz się na zrozumieniu, jak działa w kontenerze aplikacja bezstanowa lub taka, w której informacja o stanie zapisana jest na zewnątrz, zyskasz dobre podstawy do analizowania innych przypadków użycia. Bardzo polecamy rozpoczęcie od aplikacji bezstanowych i zdobycie w ten sposób pewnego doświadczenia przed przejściem do bardziej zaawansowanych zastosowań. Należy zauważyć, że społeczność intensywnie pracuje nad tym, by lepiej wspierać aplikacje stanowe w Dockerze i prawdopodobnie w ciągu najbliższego roku lub kilku lat pojawi się na tym polu wiele nowych rozwiązań.

Kontenery to nie maszyny wirtualne

Dobrym punktem wyjścia do zbudowania sobie obrazu tego, w jaki sposób można wykorzystać Dockera, jest przyjęcie, że kontenery to nie wirtualne maszyny, a bardzo lekkie opakowania przechowujące pojedyncze procesy systemu Unix. W rzeczywistości taki proces może utworzyć inne procesy, ale z drugiej strony statycznie skompilowany program mógłby stanowić całą zawartość takiego kontenera (więcej informacji na ten temat znajdziesz w punkcie „Zewnętrzne zależności" w rozdziale 7.). Kontenery są też efemeryczne: mogą się pojawiać i znikać dużo łatwiej niż maszyny wirtualne.

Maszyny wirtualne z założenia zastępują rzeczywisty sprzęt, który możesz umieścić w szafie i pozostawić tam na kilka lat. Ponieważ zastępują one prawdziwy serwer, często z założenia są utrzymywane przez długi czas. Nawet w chmurze, gdzie firmy często uruchamiają i zatrzymują maszyny wirtualne na żądanie, średni czas ich działania liczony jest zwykle w dniach. Z kolei kontener może istnieć miesiącami, ale może też zostać utworzony, wykonać przez minutę zadanie, a następnie zostać zniszczony. Wszystko to jest poprawne, z tym że jest to diametralnie inne podejście niż w przypadku typowego zastosowania maszyn wirtualnych.

Kontenery są lekkie

Szczegółami ich działania zajmiemy się później, ale powiedzmy już teraz, że na utworzenie kontenera potrzeba bardzo mało przestrzeni. Szybki test na Dockerze 1.4.1 pokazuje, że nowy kontener utworzony z istniejącego obrazu zajmuje całe 12 kilobajtów miejsca na dysku. To dość mało. Z drugiej strony nowa maszyna wirtualna utworzona z wzorcowego obrazu może potrzebować setek lub tysięcy megabajtów. Nowy kontener jest tak mały, ponieważ zawiera jedynie wskaźnik do warstwowego systemu plików i metadanych z informacjami o konfiguracji.

Lekkość kontenerów oznacza, że możesz je zastosować tam, gdzie tworzenie kolejnej maszyny wirtualnej byłoby zbyt kosztowne, lub w sytuacjach, w których potrzebujesz czegoś rzeczywiście efemerycznego. Na przykład prawdopodobnie nie uruchomiłbyś całej wirtualnej maszyny, by uruchomić z zewnętrznej maszyny zapytanie `curl` do serwisu internetowego, ale z kolei do takiego zastosowania możesz utworzyć nowy kontener.

Dążenie do niezmienności infrastruktury

Wdrażając większość aplikacji w kontenerach, można zacząć upraszczać sobie zarządzanie konfiguracją, kierując się ku niezmiennej infrastrukturze (ang. *immutable infrastructure*). Idea niezmiennej infrastruktury zyskała ostatnio popularność jako odpowiedź na rzeczywiste trudności pojawiające się przy utrzymywaniu kodu do zarządzania prawdziwie idempotentną konfiguracją. Gdy taki kod się rozrasta, może się stać w utrzymaniu równie nieporęczny i trudny jak wielkie, monolityczne, przestarzałe aplikacje. Za pomocą Dockera można tworzyć bardzo lekkie serwery, które praktycznie nie wymagają zarządzania konfiguracją, a w wielu przypadkach zupełnie jej nie potrzebują. Całe zarządzanie aplikacją polega na prostym uruchamianiu kolejnych kontenerów na serwerze. Gdy konieczna będzie aktualizacja czegoś ważnego, takiego jak demon Dockera lub jądro Linuksa, możesz po prostu uruchomić nowy serwer z wprowadzonymi zmianami, uruchomić na nim swoje kontenery, a następnie odłączyć lub zaktualizować stary serwer.

Ograniczona izolacja

Kontenery są od siebie odizolowane, ale jest to ograniczone bardziej, niż mógłbyś przypuszczać. Choć możesz ograniczyć ich zasoby, to przy domyślnej konfiguracji kontenera wszystkie one współdzielą zasoby procesora i pamięci systemu operacyjnego hosta, tak jak oczekiwałbyś tego od funkcjonujących na tym samym sprzęcie procesów Unix. Oznacza to, że dopóki nie ograniczysz kontenerów, mogą one rywalizować o zasoby na Twoich maszynach produkcyjnych. Czasem jest to sytuacja pożądana, ale ma to wpływ na podejmowanie decyzji projektowych. Możliwe jest wprowadzenie w Dockerze limitów na wykorzystanie procesora i pamięci, ale w większości sytuacji nie są one włączone domyślnie tak, jak byłyby w przypadku maszyny wirtualnej.

Często zdarza się, że wiele kontenerów dzieli jedną warstwę systemu plików lub więcej takich warstw. Jest to jedna z ważniejszych decyzji projektowych w Dockerze, jednak oznacza to też, że jeśli zaktualizujesz dzielony obraz, będziesz musiał odtworzyć wiele kontenerów.

Procesy w kontenerach są też po prostu procesami na samym serwerze Dockera. Działają one, korzystając z dokładnie tej samej instancji jądra Linuksa, co system operacyjny hosta. Pojawiają się one nawet w wynikach działania programu `ps` na serwerze, na którym działa Docker. Jest to zupełnie inaczej niż w przypadku hipernadzorcy, gdzie stopień izolacji procesów zazwyczaj obejmuje uruchomienie zupełnie oddzielnej instancji systemu operacyjnego dla każdej maszyny wirtualnej.

Takie lekkie, domyślne ograniczenia mogą kusić, by udostępniać więcej zasobów z serwera, takich jak systemy plików pozwalające na przechowywanie informacji o stanie. Powinieneś jednak mocno się zastanowić przed dalszym udostępnianiem kontenerowi zasobów z serwera, jeśli nie będą one używane wyłącznie przez ten kontener. O bezpieczeństwie kontenerów porozmawiamy później, ale generalnie powinieneś rozważać zastosowanie reguł SELinux lub AppArmor do zwiększenia ich izolacji, a nie likwidować istniejące bariery.

Domyślnie wiele kontenerów korzysta z UID 0 do uruchamiania procesów. Ponieważ kontener jest *zamknięty*, wygląda to bezpiecznie, ale w rzeczywistości tak nie jest. Ponieważ wszystko działa na tym samym jądrze, wiele rodzajów zagrożeń bezpieczeństwa lub najprostszy błąd w konfiguracji mogą dać użytkownikowi *root* kontenera nieautoryzowany dostęp do zasobów systemowych, plików i procesów hosta.

Aplikacje bezstanowe

Przykładem aplikacji, które dobrze zamyka się w kontenery, są aplikacje sieciowe przechowujące swój stan w bazach danych. Mógłbyś też uruchomić coś w stylu instancji `memcache` w kontenerach. Jeśli jednak przyjrzysz się dokładniej swojej aplikacji sieciowej, to dostrzeżesz, że ma ona prawdopodobnie lokalnie zapisane informacje o stanie, z których korzystasz — takie jak pliki konfiguracyjne. Może nie wydaje się to wiele, ale oznacza, że ograniczyłeś możliwość ponownego wykorzystania swojego kontenera i utrudniłeś jego wdrożenie w innym środowisku bez zadbania o utrzymanie poprawnych danych konfiguracyjnych w jego zasobach.

W wielu przypadkach proces zamykania aplikacji w kontenerze oznacza przeniesienie stanu konfiguracji do zmiennych środowiska, które można przekazać do aplikacji z kontenera. Umożliwia to wykorzystanie tego samego kontenera zarówno w środowisku produkcyjnym, jak i demonstracyjnym. W wielu firmach takie środowiska będą potrzebowały wielu różnych ustawień konfiguracyjnych, od nazw baz danych do nazw serwerów udostępniających inne powiązane usługi.

Podczas korzystania z kontenerów możesz też zauważyć, że zawsze zmniejszasz rozmiar swojej aplikacji podczas zamykania jej w kontener, usuwając z niej wszystko, co nie jest niezbędne do działania. Zauważyliśmy, że zastanowienie się nad tym, co jest potrzebne do działania w architekturze rozproszonej w postaci kontenera, może prowadzić do ciekawych decyzji projektowych. Jeśli na przykład masz usługę zbierającą dane, przetwarzającą je i zwracającą wynik, możesz skonfigurować kontenery na wielu serwerach do wykonywania tej pracy, a następnie łączyć wyniki ich działania w innym kontenerze.

Przenoszenie informacji o stanie na zewnątrz

Jeśli Docker najlepiej sprawdza się w przypadku aplikacji bezstanowych, to w jaki sposób najlepiej zapisywać stan, gdy jest to konieczne? Konfigurację najlepiej przekazywać na przykład przez zmienne środowiska. Docker obsługuje zmienne środowiska — są one przechowywane w metadanych opisujących konfigurację kontenera. Oznacza to, że przy restartowaniu kontenera za każdym razem do aplikacji będzie przekazywana taka sama konfiguracja.

To w bazach danych skalowalne aplikacje często przechowują informacje o stanie. Nic nie stoi na przeszkodzie, by w przypadku Dockera robić to samo w aplikacjach zamkniętych w kontenerze. W przypadku aplikacji, które muszą zapisywać pliki, trzeba jednak zmierzyć się z pewnymi wyzwaniami. Zapis informacji w systemie plików kontenera nie będzie dobrze działać, ponieważ dostępna będzie bardzo ograniczona przestrzeń i nie będzie zachowywany stan pomiędzy restartami kontenera. Przed umieszczeniem w kontenerze Dockera aplikacji, które muszą zapisywać stan systemu plików, należy się mocno zastanowić. Jeśli zdecydujesz, że skorzystanie z Dockera w takich sytuacjach może przynieść korzyści, najlepiej zaprojektować rozwiązanie, w którym stan może być zapisywany w scentralizowanej lokalizacji dostępnej niezależnie od tego, na którym serwerze zostanie uruchomiony kontener. W pewnych przypadkach może to oznaczać usługi takie jak Amazon S3, RiakCS, OpenStack Swift, lokalne urządzenie blokowe czy nawet zamontowanie dysków iSCSI w kontenerze.

Choć możliwe jest zapisanie stanu w podłączonym zewnętrznym systemie plików, jest to w zasadzie odradzane przez społeczność i powinno być traktowane jako zaawansowany przypadek użycia. Bardzo zaleca się rozpoczęcie pracy z aplikacjami, które nie muszą zapisywać informacji o stanie. Istnieje wiele powodów, dla których takie działanie jest generalnie odradzane — w prawie wszystkich przypadkach przyczyną jest to, że w ten sposób wprowadzane są zależności pomiędzy kontenerem a hostem, które przeszkadzają w wykorzystaniu Dockera jako prawdziwie dynamicznej, horyzontalnie skalowalnej usługi umożliwiającej dostarczanie aplikacji. Jeśli Twój kontener korzysta z podłączonego systemu plików, może być uruchomiony jedynie w systemie zawierającym ten system plików.

Schemat pracy z Dockerem

Tak jak w przypadku wielu innych narzędzi, podczas korzystania z Dockera bardzo mocno zaleca się stosowanie opracowanego dla niego schematu pracy. Jest to bardzo użyteczny schemat, dobrze odzwierciedlający sposób organizacji wielu firm, ale prawdopodobnie różni się on odrobinę od tego, co Ty lub Twój zespół stosujecie w tej chwili. Po przekształceniu sposobu pracy w naszej organizacji do postaci stosowanej w Dockerze możemy z całą pewnością powiedzieć, że ta zmiana wpłynęła korzystnie na wiele zespołów w organizacji. Jeśli ten schemat pracy zostanie dobrze zaimplementowany, może rzeczywiście pomóc zrealizować obietnicę zredukowania zakresu niezbędnej komunikacji między zespołami.

Wersjonowanie

Pierwszą rzeczą, którą Docker domyślnie oferuje, jest mechanizm kontroli wersji w dwóch postaciach. Pierwszy mechanizm jest wykorzystywany do śledzenia warstw systemu plików, z których to warstw są zbudowane obrazy, a drugi to system znakowania tworzonych kontenerów.

Warstwy systemu plików

Kontenery Dockera składają się z ułożonych na stosie warstw systemu plików, z których każda jest identyfikowana za pomocą unikalnego ciągu znaków, przy czym każdy nowy zestaw zmian wprowadzany podczas tworzenia obrazu umieszczany jest nad wszystkimi poprzednimi zmianami. Wspaniale się to sprawdza, ponieważ oznacza to, że gdy tworzysz nową kompilację, musisz przebudować jedynie te warstwy, w których zmiana jest wprowadzana, oraz te, które na nich się opierają. Oszczędza to czas i łącza sieciowe, ponieważ kontenery dostarczane są w postaci warstw i nie ma konieczności dostarczania tych warstw, które serwer już ma zapisane. Jeśli wykonywałeś już wdrożenia za pomocą innych narzędzi do wdrażania, to wiesz, że może się to skończyć wielokrotnym przesyłaniem setek megabajtów tych samych danych na serwer przy każdym wdrożeniu. Działa to wolno, a co gorsza, nie możesz tak naprawdę być pewien, co zmieniło się pomiędzy kolejnymi wdrożeniami. Dzięki efektowi warstwowania oraz dzięki temu, że kontenery Dockera zawierają wszystkie zależności aplikacji, możesz być praktycznie pewien, gdzie nastąpiła zmiana.

Aby to trochę uprościć, musisz pamiętać, że obraz w Dockerze zawiera wszystko, co jest potrzebne Twojej aplikacji. Jeśli zmienisz jeden wiersz kodu, na pewno nie będziesz chciał tracić czasu na przebudowywanie wszystkich bibliotek wykorzystywanych przez Twój kod, by utworzyć nowy obraz. Zamiast tego Docker wykorzysta tak wiele warstw, jak to możliwe, aby zostały przebudowane tylko warstwy, na które wpłynęła zmiana w kodzie.

Znaczniki obrazów

Druga postać kontroli wersji oferowana przez Dockera to ta, która ułatwia odpowiedź na ważne pytanie: jaka była poprzednia wersja wdrożonej aplikacji? Odpowiedź nie zawsze jest prosta. Istnieje wiele rozwiązań dla aplikacji niekorzystających z Dockera — od znaczników Gita dla każdej wersji, przez logi wdrożenia, po znakowanie kompilacji przeznaczonych do wdrożenia i wiele więcej. Jeśli na przykład koordynujesz wdrożenie za pomocą Capistrano, zajmie się ono tym za Ciebie, przechowując na serwerze zdefiniowaną liczbę wcześniejszych wersji i wykorzystując linki symboliczne, by uczynić jedną z nich wersją bieżącą.

Jednak w każdym skalowalnym środowisku produkcyjnym każda aplikacja ma swój unikalny sposób przechowywania informacji o wersjach wdrażanego kodu. Może być też tak, że wiele aplikacji działa podobnie, a jedna się wyróżnia. Gorzej, że w środowisku korzystającym z wielu języków programowania wykorzystywane są przez aplikacje różne narzędzia do wdrażania, często mające ze sobą bardzo mało wspólnego. Zatem pytanie „jaka była poprzednia wersja?" może mieć wiele odpowiedzi w zależności od tego, kogo spytasz i jakiej aplikacji będzie dotyczyło pytanie. Docker ma do obsługi tego wbudowany mechanizm, który umożliwia znakowanie obrazów podczas wdrażania. Możesz pozostawić wiele wersji swojej aplikacji na serwerze i po prostu oznaczać je podczas udostępniania. Jak już wspomnieliśmy, nie jest to skomplikowane ani też trudne do znalezienia w innych narzędziach do wdrażania oprogramowania. Możesz wprowadzić oznaczanie wersji podczas udostępniania jako standard w swoich aplikacjach i wówczas wszyscy będą mogli oczekiwać tego, że oznaczenia będą stosowane w taki sam sposób.

W wielu przykładach, jakie można znaleźć w internecie oraz w tej książce, zobaczysz, że użytkownicy korzystają ze znacznika `latest` (najnowsze). Jest to znacznik przydatny na początek oraz podczas pisania przykładów, ponieważ w ten sposób zawsze wybieramy najnowszą wersję obrazu. Jest to jednak znacznik dynamiczny, więc nie jest dobrym pomysłem stosowanie go w większości praktycznych zastosowań, gdyż wykorzystywane biblioteki mogą zostać zaktualizowane niezależnie od Ciebie i nie będzie możliwe przywrócenie poprzedniej wersji, ponieważ znacznik `latest` nie będzie wskazywał już na tę samą wersję.

Budowanie

Budowanie aplikacji w wielu organizacjach jest czarną magią i tylko kilka osób wie, które dźwignie i gałki trzeba poruszyć, by uzyskać poprawny, możliwy do wdrożenia artefakt. Część kosztów związanych z wdrożeniem nowej aplikacji jest związana z poprawnym jej zbudowaniem. Docker nie rozwiązuje wszystkich problemów, ale dostarcza narzędzia z ustandaryzowaną konfiguracją oraz zestaw narzędzi do budowania. Bardzo ułatwia to użytkownikom nauczenie się poprawnego budowania Twojej aplikacji oraz uruchamiania utworzonego kodu.

Narzędzie Dockera działające w trybie tekstowym zawiera parametr `build`, który pozwala na pobranie pliku *Dockerfile* i utworzenie obrazu Dockera. Każde polecenie w *Dockerfile* generuje nową warstwę w obrazie, dzięki czemu, po prostu zaglądając do samego pliku *Dockerfile*, można łatwo ustalić, jakie operacje będą wykonywane podczas budowania. Dobrą stroną całej tej standaryzacji jest to, że każdy inżynier, który pracował z *Dockerfile*, może tam zajrzeć i zmodyfikować proces budowania każdej innej aplikacji. Ponieważ obraz Dockera jest artefaktem ustandaryzowanym, wszystkie narzędzia stosowane w procesie budowania będą takie same niezależnie od wykorzystanego narzędzia, dystrybucji systemu operacyjnego, w którym on działa, czy liczby niezbędnych warstw.

Większość procesów budowania w Dockerze składa się z jednego polecenia `docker build` i generuje pojedynczy artefakt, obraz kontenera. Ponieważ jest to zazwyczaj przypadek, w którym większość logiki związanej z budowaniem jest w całości zamknięta w pliku *Dockerfile*, łatwo utworzyć standardowe zadania budowania dla wszystkich zespołów do wykorzystania w systemach budowania takich jak Jenkins (*http://jenkins-ci.org/*). Dla dokładniejszego ustandaryzowania procesu budowania kilka firm, w tym eBay, rzeczywiście przyjęło za standard kontenery Dockera, by budować obrazy z pliku *Dockerfile*.

Testowanie

Choć sam Docker nie zawiera wbudowanego frameworka do testowania, to sposób, w jaki kontenery są budowane, daje kilka korzyści podczas testowania kodu zamkniętego w kontenerach Dockera.

Testowanie aplikacji produkcyjnej może przyjmować wiele postaci: od testów jednostkowych do pełnych testów integracyjnych w środowisku zbliżonym do produkcyjnego. Docker ułatwia testowanie, zapewniając, że do produkcji zostaną dostarczone te same artefakty, które przeszły testowanie. Można to zagwarantować, ponieważ by upewnić się, że dostarczamy tę samą wersję aplikacji, możemy wykorzystać zarówno obliczoną przez Dockera sumę kontrolną SHA dla kontenera, jak i dodatkowy znacznik.

Druga cecha istotna z punktu widzenia testowania jest taka, że wszystkie testy kontenera automatycznie obejmują testowanie aplikacji ze wszystkimi wykorzystywanymi przez nią bibliotekami, z którymi jest ona dostarczana. Jeśli framework do testów jednostkowych zwróci informację, że wszystkie testy obrazu kontenera się powiodły, możesz mieć pewność, że nie doświadczysz na przykład problemów z wersjami bibliotek podczas wdrażania. Nie jest to proste do osiągnięcia w przypadku innych technologii i przykładowo nawet testy plików WAR Javy nie obejmują testowania samego serwera aplikacji. Ta sama aplikacja Java wdrażana w kontenerze Dockera będzie też zawierała serwer aplikacji, dzięki czemu cały stos może zostać dokładnie przetestowany przed wdrożeniem produkcyjnym.

Dodatkową korzyścią z dostarczania aplikacji w kontenerach Dockera jest to, że w miejscach, gdzie wiele aplikacji komunikuje się ze sobą zdalnie za pomocą czegoś w rodzaju API, programiści jednej aplikacji mogą w prosty sposób pracować z wersją innej usługi, która jest przygotowana dla środowiska potrzebnego im w danej chwili, na przykład produkcyjnego lub demonstracyjnego. Programiści w żadnym z zespołów nie muszą być ekspertami w dziedzinie budowy czy wdrażania innej usługi — po prostu skupiają się na tworzeniu swojej aplikacji. Jeśli zastosujesz to w przypadku architektury SOA z niezliczoną ilością mikrousług, to kontenery Dockera mogą się okazać prawdziwym kołem ratunkowym dla programistów i inżynierów odpowiedzialnych za zapewnienie jakości, którzy muszą się przedzierać przez gąszcz wywołań API pomiędzy mikrousługami.

Tworzenie pakietów

Podstawowym zadaniem i głównym celem podczas korzystania z Dockera jest utworzenie pojedynczego artefaktu przy każdym budowaniu kontenera. Nie ma znaczenia, w jakim języku jest napisana Twoja aplikacja ani w jakiej dystrybucji Linuksa będziesz ją uruchamiał — za każdym razem po zakończeniu budowania kontenera otrzymasz wielowarstwowy obraz Dockera. Wszystko to jest budowane i obsługiwane za pomocą narzędzi Dockera. To właśnie z porównania do kontenera transportowego pochodzi nazwa Dockera: pojedynczej, łatwej do transportowania rzeczy, którą mogą obsługiwać uniwersalne narzędzia niezależnie od tego, co w sobie zawiera. Tak jak w przypadku portu kontenerowego czy innego punktu przeładunkowego, narzędzia Twojego Dockera będą miały styczność tylko z jednym rodzajem paczek: obrazami Dockera. Jest to ogromna zaleta, ponieważ bardzo ułatwia ponowne wykorzystanie narzędzi w pracy nad różnymi aplikacjami i oznacza, że gotowe narzędzia przygotowane przez innych będą obsługiwały zbudowane

przez Ciebie obrazy. Aplikacje, których skonfigurowanie zazwyczaj wymaga dużo wysiłku podczas wdrażania na nowym komputerze lub w systemie programistycznym, stają się z Dockerem niewiarygodnie przenośne. Po zbudowaniu kontenera może on zostać łatwo wdrożony w dowolnym systemie z uruchomionym serwerem Dockera.

Wdrażanie

Wdrożenia są obsługiwane przez tak wiele rodzajów narzędzi w wielu różnych zestawieniach, że nie byłoby możliwe wymienienie tutaj ich wszystkich. Niektóre z tych mechanizmów korzystają ze skryptów powłoki, programów takich jak Capistrano, Fabric, Ansible lub innych narzędzi utworzonych własnymi siłami. Z naszego doświadczenia zdobytego w organizacjach składających się z wielu zespołów wynika, że zazwyczaj jedna lub dwie osoby w każdym zespole znają magiczne zaklęcia pozwalające wykonać poprawne wdrożenie. Gdy coś pójdzie nie tak, cały zespół liczy na to, że przywrócą one poprawne działanie. Jak już prawdopodobnie się domyślasz, Docker usuwa większość tego typu problemów. Wbudowane narzędzia wspomagają proste, jednowierszowe strategie wdrażania pozwalające pobrać zbudowany obraz na docelowy komputer i go uruchomić. Standardowy klient Dockera obsługuje jedynie wdrożenia na pojedynczych komputerach, ale dostępne są też inne narzędzia, które ułatwiają wdrażanie na klastrach komputerów z Dockerem. Dzięki standaryzacji wprowadzanej przez Dockera zbudowany przez Ciebie obraz może być wdrożony w dowolnym systemie przy niewielkim stopniu złożoności z punktu widzenia zespołów programistycznych.

Ekosystem Dockera

Wokół Dockera tworzy się duża społeczność, której siłą napędową są programiści i administratorzy systemów. Tak jak w przypadku ruchu DevOps, spowodowało to powstanie lepszych narzędzi dzięki dostosowaniu kodu do problemów występujących podczas utrzymywania i konserwacji aplikacji. W obszary, gdzie narzędzia dostarczane przez Dockera mają luki, wkroczyły inne firmy i indywidualni programiści. Wiele z tych narzędzi również udostępniono na otwartych licencjach. Oznacza to, że każda firma może je rozszerzać i modyfikować, by dostosować je do swoich potrzeb.

Docker to komercyjna firma, która wspaniałomyślnie przekazała społeczności *open source* większość kodów źródłowych rdzenia Dockera. Firmy są gorąco zachęcane do tego, by dołączyć do społeczności i wspomagać wynikami swojej pracy zasoby *open source*. Jeśli szukasz wspieranych wersji podstawowych narzędzi Dockera, to powinieneś wiedzieć, że więcej informacji na temat ich oferty możesz znaleźć na stronie *https://www.docker.com/support*.

Koordynacja

Pierwszą ważną kategorią narzędzi rozszerzających podstawowe funkcje Dockera są programy do koordynacji i przeprowadzania masowych wdrożeń, takie jak Swarm (*https://github.com/docker/swarm/*), Centurion firmy New Relic (*https://github.com/newrelic/centurion/*) czy Helios firmy Spotify (*https://github.com/spotify/helios*). We wszystkich nich stosowane jest dość proste podejście do tematu koordynacji. W przypadku środowisk bardziej złożonych lepszym rozwiązaniem są

Kubernetes firmy Google (*https://github.com/GoogleCloudPlatform/kubernetes*) i Apache Mesos (*http://mesos.apache.org/*). Wraz z odkrywaniem kolejnych luk i publikowaniem ulepszeń przez nowych użytkowników stale pojawiają się nowe narzędzia.

Niepodzielne maszyny

Dodatkową koncepcję, którą można wykorzystać do poprawienia komfortu korzystania z Dockera, stanowią niepodzielne maszyny (ang. *atomic hosts*). Standardowo serwery i maszyny wirtualne są systemami, które organizacja starannie kompletuje, konfiguruje i utrzymuje, aby dostarczały wielu funkcji, które można wykorzystać w różnorodny sposób. Aktualizacje muszą być często przeprowadzane w wielu krokach i na wiele sposobów konfiguracja może utracić spójność oraz doprowadzić do nieoczekiwanego zachowania systemu. Większość działających w dzisiejszym świecie systemów łata się i aktualizuje w czasie rzeczywistym. W odróżnieniu od tego w świecie zorientowanym na wdrożenia większość osób wdraża pełną, spójną wersję swojej aplikacji, zamiast próbować dokładać łaty do działającego systemu. Jedną z zachęcających cech kontenerów jest to, że pomagają one uczynić aplikacje jeszcze bardziej spójnymi, niż ma to miejsce w przypadku modeli tradycyjnych wdrożeń.

Co by było, gdybyś mógł rozszerzyć ten podstawowy wzorzec kontenerów na cały system operacyjny? Zamiast polegać na zarządzaniu konfiguracją i próbować aktualizować, łatać komponenty Twojego systemu operacyjnego i go modyfikować, mógłbyś po prostu wyciągnąć nowy, spójny obraz systemu operacyjnego i zrestartować serwer? A gdy coś się zepsuje, mógłbyś łatwo wrócić do tego samego obrazu, jaki wykorzystywałeś poprzednio?

Jest to jedna z podstawowych koncepcji stojących za opartymi na Linuksie dystrybucjami do obsługi pojedynczych komputerów, takimi jak CoreOS (*https://coreos.com/*) oraz Project Atomic (*http://www.projectatomic.io/*). Nie tylko możesz łatwo usuwać i ponownie wdrażać swoje aplikacje, ale tę samą filozofię możesz zastosować do całego stosu oprogramowania. Ten wzorzec pomaga wprowadzić spójność na niesamowitym poziomie i elastyczność całego stosu.

Typowe cechy niepodzielnych maszyn (*https://gist.github.com/jzb/0f336c6f23a0ba145b0a*) to minimalna wielkość, architektura ukierunkowana na obsługę kontenerów i Dockera oraz możliwość przeprowadzania aktualizacji i ich wycofywania w taki sposób, by było je łatwo kontrolować za pomocą narzędzi do koordynacji wielu maszyn działających zarówno na sprzęcie fizycznym, jak i na popularnych platformach do wirtualizacji.

W rozdziale 3. omówimy, w jaki sposób możesz łatwo wykorzystać niepodzielne maszyny w swoim procesie wytwarzania oprogramowania. Jeśli korzystasz z niepodzielnych maszyn, przeprowadzając wdrożenia, to dzięki temu możesz uzyskać niespotykaną wcześniej spójność stosu oprogramowania w środowiskach programistycznych i produkcyjnych.

Dodatkowe narzędzia

Pozostałe kategorie obejmują narzędzia do audytu, logowania, zarządzania siecią, mapowania i wiele innych, z których większość korzysta bezpośrednio z API Dockera. Wśród takich narzędzi i mechanizmów powiązanych z Dockerem można wymienić Flannel dla Kubernetesa z CoreOS,

Weave, wirutalną sieć pozwalającą porozmieszczać kontenery na wielu maszynach z Dockerem oraz bezpośrednie wsparcie dla logów Dockera w ruterze logów Heka Mozilli (*https://github.com/mozilla-services/heka*).

Nikt się nie spodziewał tak dużej społeczności dynamicznie rozwijającej się wokół Dockera, ale bardzo prawdopodobne, że takie wsparcie jedynie przyspiesza rozwój zastosowań Dockera i tworzenie solidnych narzędzi rozwiązujących wiele problemów pojawiających się w społeczności.

Podsumowanie

W ten sposób skończyliśmy szybkie omówienie Dockera. Wrócimy do tej rozmowy później, by odrobinę dokładniej przyjrzeć się architekturze Dockera, podać więcej przykładów zastosowania narzędzi tworzonych przez społeczność oraz głębiej zanurzyć się w rozważania niezbędne do projektowania solidnych platform kontenerowych. Na razie jednak prawdopodobnie masz wielką ochotę, by tego wszystkiego spróbować, dlatego więc w kolejnym rozdziale zajmiemy się instalacją i uruchomieniem Dockera.

Skorowidz

A

adres MAC, 84
aktualizacja obrazów, 101
Amazon EC2 Container Service, 148
API, Application Programming Interface, 30
aplikacje bezstanowe, 34
architektura, 28
AUFS, 164
automatyczne restartowanie kontenera, 93
autoryzacja w rejestrze, 71
AWS CLI, 149

B

badanie powłoki, 107
bezpieczeństwo, 173
Boot2Docker, 53
BTRFS, 164
budowanie, 37, 190
    obrazu, 64

C

chmura, 23
CLI, Command Line Interface, 149
czyszczenie obrazów, 96

D

debugowanie kontenerów, 129
definicja zadania, 153
demon
    Dockera, 75, 178
    upstart, 49
devicemapper, 164
DNS, 84
Docker, 19, 99
    Machine, 50
    Swarm, 140
dokument
    The Reactive Manifesto, 194
    The Twelve-Factor App, 194
dostarczanie obrazów, 77
dyspozycyjność, 192

E

EC2 Container Service, 148
ekosystem, 39
elastyczność, 194

F

Fedora Linux 21, 46
framework do wdrażania, 23

G

gniazdka sieciowe, 29
grupy kontrolne, 166
GUI, 47

H

historia
    kontenerów, 80
    obrazów, 136
hostname, 83

I

IAM, Identity and Access Management, 148
informacje
  o kontenerze, 102
  o serwerze, 100
  o stanie, 35
instalacja, 43–59
  z Homebrew, 47
instalator z GUI, 47
  klienta Netcat, 157
  klienta Telnet, 157
instancje kontenerów, 150
izolacja, 34
izolowanie zależności, 186

J

jawne deklarowanie, 186

K

klient, 43, 44
  Netcat, 157
  Telnet, 157
konfiguracja
  demona Dockera, 75
  IAM, 148
  kontenera, 82
kontenery, 31, 44, 79
  adres MAC, 84
  automatyczne restartowanie, 93
  bezpieczeństwo, 173
  czyszczenie, 96
  debugowanie, 129
  dodatkowe uprawnienia, 175
  konfiguracja, 82
  pauzowanie, 96
  pobieranie informacji, 102
  produkcyjne, 121
  projektowanie platformy produkcyjnej, 185
  przeglądanie, 136
  testowanie, 125
  tworzenie, 81
  uruchamianie, 92
  wnętrze, 103
  wymuszanie zakończenia pracy, 95
  wznawianie pracy, 96
  zatrzymywanie, 94
konto w Docker Hub, 72
kontrolowanie procesów, 134
koordynacja, 39, 123
kopia rejestru, 74, 76

L

Linux, 45
logi, 109, 193
logowanie do rejestru, 72

Ł

łączenie z kontenerem, 158

M

Mac OS X 10.10, 47
magazyny danych, 85, 163
maszyny
  niepodzielne, 40
  wirtualne, 33, 50
mechanizm devicemapper, 164
mechanizmy wymienne, 161
menedżer Swarm, 142
Microsoft Windows 8, 48
minimalizowanie liczby obiektów, 198
model klient-serwer, 28
monitorowanie Dockera, 112

N

narzędzia, 122
  do koordynacji, 123
  do planowania przetwarzania rozproszonego, 123
  dodatkowe, 40
  rozbudowane, 29
nazwa kontenera, 82
niepodzielna maszyna, 44
niezmienność infrastruktury, 33

O

obiekty do wdrożenia, 198
obrazy, 36, 44, 61
  bazowe, 69
  budowanie, 64
  historia, 136

pobieranie aktualizacji, 101
uruchamianie, 68
zapisywanie, 70
obsługa komunikatów, 195
ograniczenia zasobów, 87
optymalizacja
przechowywania danych, 199
przesyłania danych, 199
overlayfs, 165

P

pakiet util-linux, 104
pakiety, 38
pamięć, 90
para klucz-wartość, 82
pauzowanie pracy kontenera, 96
platformy do wirtualizacji, 23
plik
.dockerignore, 65
Dockerfile, 61
index.js, 69
pobieranie aktualizacji obrazów, 101
polecenie
bind mount, 83
docker exec, 103
docker kill, 95
docker top, 129
systemd, 49
ulimit, 91
porty sieciowe, 29, 191
powłoka, 107
proces wdrożenia, 27
procesor, 87
procesy, 21, 129, 190
administracyjne, 193
generowane dane, 129
kontrolowanie, 134
przeglądanie, 133
program
Boot2Docker, 53
cAdvisor, 117
Centurion, 144
CMD, 157
docker events, 115
nsenter, 104
Vagrant, 55
projekt LXC, 162
projektowanie produkcyjnej platformy, 185
przechowywanie kopii rejestru, 76
przeglądanie
kontenera, 136
procesów, 133
przestrzeni nazw, 171
sieci, 135
systemu plików, 138
przenoszenie informacji, 35
przepływ pracy, 198
przestrzenie nazw
AppArmor, 177
IPC, 170
jądra, 169
montowania, 170
PID, 170
SELinux, 177
sieciowe, 171
UTS, 170
użytkownika, 169, 171
przesyłanie danych, 199
przetwarzanie rozproszone, 123
przypinanie procesora, 89
publiczne rejestry, 70

R

rejestr obrazów, 70
rejestry prywatne, 71
repozytorium, 73
kodów, 186
responsywność, 194

S

SaaS, Software-as-a-Service, 186
schemat pracy, 36
serwer, 43, 48
serwery na maszynach wirtualnych, 50
sieć, 180
w kontenerze, 31
skalowanie, 139
stabilność, 194
standardowe wyjście błędów stderr, 109
statystyki kontenerów, 112
sterowniki
magazynów danych, 165
uruchamiania, 161
vfs, 165
stosowanie procesów, 21

strumienie zdarzeń, 193
system init.d, 49
system plików, 36, 138
  /sys, 167
  btrfs, 164
  overlayfs, 165
szybki przegląd, 125

Ś

środowisko
  produkcyjne, 185, 192
  programistyczne, 24, 192

T

tekstowy klient Dockera, 30
testowanie, 38, 58
  kontenerów, 125
  lokalnej kopii rejestru, 76
  zadania, 157
tworzenie
  kontenera, 81
  kopii rejestru, 74
  pakietów, 38

U

Ubuntu Linux, 45
udostępnianie, 190
udziały procesora, 87
UID 0, 173
upraszczanie procesów, 25
uruchamianie, 190
  kontenera, 92
  obrazu, 68
urządzenia blokowe, 91
usługa
  EC2 Container Service, 148
  przechowująca kopię rejestru, 76
usługi pomocnicze, 190

V

Vagrant, 55
vfs, 165

W

warstwy systemu plików, 36
wdrażanie, 39, 121
wersje Dockera, 99
wersjonowanie, 36
wirtualizacja, 23
własne obrazy bazowe, 69
wsparcie, 27
współbieżność, 191
wykorzystanie portów, 191
wyświetlanie wersji, 99
wznawianie pracy kontenera, 96

Z

zadania, 153
zależności, 186
zapisywanie
  konfiguracji, 188
  obrazów, 70
zarządzanie
  klastrem, 142
  konfiguracją, 23
  obciążeniem, 24
zastosowania Dockera, 32
zatrzymywanie
  kontenera, 94
  zadania, 158
zewnętrzne zależności, 128
zmienna $WHO, 69
zmienne środowiska, 69
znaczniki obrazów, 36
zwracanie wyniku, 107

O'REILLY®

Docker — sposób na niezawodne zarządzanie rozwojem aplikacji!

Docker został zaprezentowany światu w marcu 2013 roku i praktycznie od początku wzbudza zaskakujące zainteresowanie. Narzędzie to pozwala na proste zarządzanie procesem tworzenia określonego elementu aplikacji, wdrażania go na dużą skalę w dowolnym środowisku oraz usprawniania przepływu pracy. Ma przy tym duże możliwości i łączy w sobie prostotę wdrażania aplikacji z prostotą administrowania. Jednym słowem, Docker jest niezwykle użytecznym narzędziem!

Niniejsza książka jest praktycznym przewodnikiem, dzięki któremu Docker przyczyni się do sukcesu organizacji na wiele sposobów: uprości podejmowanie decyzji dotyczących architektury, ułatwi pisanie narzędzi pomocniczych, a przede wszystkim umożliwi bezproblemowe przeprowadzanie integracji kolejnych elementów aplikacji. Opisano tu, w jaki sposób za pomocą Dockera można przygotować pakiet aplikacji ze wszystkimi ich zależnościami, a następnie je testować, wdrażać, skalować oraz utrzymywać ich pracę w środowiskach produkcyjnych.

W tej książce omówiono:

- wykorzystanie Dockera do automatyzacji i uproszczenia obsługi pakietów
- zasady pracy z obrazami, kontenerami i aplikacjami Dockera
- dołączanie do kodu aplikacji niezbędnych plików systemu operacyjnego
- możliwość testowania tego samego elementu aplikacji we wszystkich systemach i środowiskach

Sean P. Kane — jest starszym inżynierem w firmie New Relic. Ma wieloletnie doświadczenie w utrzymywaniu aplikacji produkcyjnych w rozmaitych dziedzinach przemysłu. Zajmował się kwestiami bezpieczeństwa systemów oraz automatyzacją sprzętową. Mieszka z rodziną na północno-zachodnim wybrzeżu Stanów Zjednoczonych, lubi podróżować i fotografować.

Karl Matthias — jest głównym inżynierem systemowym w firmie Nitro Software. Pracował jako programista, administrator systemów oraz inżynier sieciowy w różnych firmach. Mieszka w Dublinie w Irlandii. Lubi spędzać czas z rodziną, kręcić filmy starymi kamerami i jeździć na rowerze.

ISBN 978-83-283-2904-1

9 788328 329041

Informatyka w najlepszym wydaniu

cena: 44,90 zł